Chapter FIVE Enumerations

Exam Objectives

Use enumerated types including methods, and constructors in an enum type.

Enumerations

Let's say our application uses three states or values for the volume of playing a song: high, medium, low. We typically model this with some "constants", like this:

```
public interface Volume {
    public static final int HIGH = 100;
    public static final int MEDIUM = 50;
    public static final int LOW = 20;
}
```

However, this is not a good implementation. Consider a method to change the volume:

```
public void changeVolumen(int volume) {
    ...
}
...
app.changeVolumen(Volume.HIGH);
```

What is stopping someone to call this method with an arbitrary value like:

```
app.changeVolume(10000);
```

Of course, we can implement some checks, but this just makes the problem more complicated. Luckily, enumerations (or enums for short) provide a nice solution to this issue.

An *enum* is a type (like a class or an interface) that represents a **FIXED** list of values (you can think of these as constants).

So we can use an enum to represent the volume of a sound in our application (notice the use of `enum` instead of `interface`):

```
public enum Volume {
    HIGH, MEDIUM, LOW
}
```

Notice that since the values are "constants" (they are implicitly `public` , `static` , and `final`) the convention of using all caps is followed.

Also, notice that the values are comma-separated and because the `enum` only contains a list of values, the semicolon is optional after the last one.

This way, we can define the `changeVolume` method as:

```
public void changeVolumen(Volume volume) {
    ...
}
```

Passing any other object that is not a `Volume` , will generate a compile-time error:

```
changeVolume(Volume.HIGH); // All good
changeVolume(-1); // Compile-time error
```

The method is now type-safe, meaning that we can't assign invalid values.

By the way, because of something we review later, you cannot use the new operator to get a reference of an enum, so we just get the reference directly:

```
Volume level = Volume.LOW;
```

You can also get an enum from a string, for example:

```
Volume level = Volume.valueOf("LOW");
```

Just be careful, this method is case sensitive:

```
Volume level = Volume.valueOf("Low"); // Run-time exception
```

To get all the enums of a particular type use the method `values()` , that returns an array of enums in the same order in which they were declared and that pairs great with a `for-each` statement:

```
for(Volume v: Volume.values()) {
   System.out.print(v.name() + ", ");
}
```

The output:

```
HIGH, MEDIUM, LOW,
```

Working with Enums

In the first example, `HIGH` was equal to `100` . But now that we are using enums, what is the value of `HIGH` ?

If we print its value:

```
System.out.println(Volume.HIGH);
```

The output will be the name of the enum:

```
HIGH
```

This is equivalent to invoke the `name()` method that all enums have:

```
System.out.println(Volume.HIGH.name());
```

In other words, the `toString()` implementation of an enum calls the `name()` method.

Enums also have a zero-based value that corresponds to the order in which they're declared. You can get it with the `ordinal()` method:

```
System.out.println(Volume.HIGH.ordinal());
System.out.println(Volume.LOW.ordinal());
```

The output will be:

```
0
2
```

But this might not be enough for all cases (like in our example). So we can define a constructor (that is called the first time the `enum` is used) that accepts a parameter that will be stored in an instance variable:

```
public enum Volume {
    HIGH(100), MEDIUM(50), LOW(20);
    private int value;

    private Volume(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
```

The only restriction is the constructor must be `private`. Otherwise, the compiler will throw an error. Notice that a getter method was also added.

We can add a setter method also, but generally, it is not needed since enums work more like constants. However, keep in mind that changing the value of the instance variable is allowed, while reassigning the `enum` is not because they are implicitly `final` and cannot be changed after their creation:

```
// Compile-time error
Volume.HIGH = Volume.MEDIUM;
```

Talking about compile-time errors, the following statements will also generate one:

```
// Use Volume.HIGH.ordinal()
if(Volume.HIGH == 0) {
    ...
}
// Use Volume.HIGH.getValue()
if(Volume.HIGH == 100) {
    ...
}
```

```
// A enum can't extend a class
public enum Volume extends AClass { ... }
```

To compare the value of an `enum` you can use the either the ordinal or name methods, in addition to any other custom method. You can also compare an `enum` to another `enum` with either the `==` operator (because enums are `final`), the equals method, or by using a `switch` statement:

```
Volume level = Volume.HIGH;
...
// Or Volume.HIGH.equals(level)
if(Volume.HIGH == level) {
    ...
}
switch(level) {
    // Notice that the only the name of the enum is used,
    // in fact, Volume.HIGH for example, won't compile
    case HIGH: ...
    case MEDIUM: ...
    case LOW: ...
}
```

An `enum` type **CANNOT** extend from a class because implicitly, all enums extend from `java.lang.Enum` . What you **CAN** do is implement interfaces:

```
public enum Volume implements AnInterface { ... }
```

The closest we can get to extending a class when working with enums is overriding methods and implementing abstract methods. For example:

```
public enum Volume {
    HIGH(100) {
       public void printValue() {
           System.out.println("** Highest value**");
       }
       public void printDescription() {
           System.out.println("High Volume");
       }
    }, MEDIUM(50) {
        public void printDescription() {
           System.out.println("Medium Volume");
        }
    }, LOW(20) {
        public void printDescription() {
            System.out.println("Low Volume");
        }
    };
    private int value;

    private Volume(int value) {
        this.value = value;
    }

    public void printValue() {
        System.out.println(value);
    }
    public abstract void printDescription();
}
```

In the case of `printValue()` , `MEDIUM` and `LOW` will use the enum-level version that just prints the value while `HIGH` will use its own version. If the method is abstract, every `enum` has to implement it. Otherwise, a compile-time error will be thrown.

Enums as Singletons

Remember back in Chapter 1 when I mention that enums are singletons? Do you know why now?

- You cannot create an instance of an enum by using the new operator (because the constructor is `private`).
- An instance of an enum is created when the enum is first referenced.
- An enum can't be reassigned.
- I didn't mention it before, but enums are thread-safe by default (meaning that you don't need to do double checks when creating them).
- In Chapter 1, the impact of serialization on singletons wasn't really explored, but if you serialize a singleton, when you get it back with the default implementation of `readObject()`, this method will always return a new instance, so the singleton is not really one anymore. However, when serializing an `enum`, this won't happen.

Given those reasons, most people believe that most of the time, enums are the best way to implement the singleton design pattern in Java.

Key Points

- An *enum* is a type that represents a **FIXED** list of values (you can think of these as constants), providing type safety.
- Enums can define constructors, but they must be `private`. Otherwise, a compile-time error will be thrown.
- Enums are implicitly `public`, `static` and `final`.
- An `enum` can be created from a String using the case-sensitive `valueOf()` method.
- To get all the enums of a certain type use the method `values()`, that returns an array of enums in the same order in which they were declared.
- When the `toString()` method is invoked, it prints the name of the enumeration.
- Enums can be compared against other enums using the `==` operator and the `equals()` method.
- Enums can be used in `switch` statements.
- Enums can implement interfaces, but they cannot extend from a class since they implicitly extend from `java.lang.Enum`.
- Enums are the easiest way to implement singletons.

Self Test

1. Given:

```
public enum Question_5_1 {
    UP(1) {
        public void printValue() {
            System.out.println(value);
        }
    }, DOWN(0);
    private int value;

    private Question_5_1(int value) {
        this.value = value;
    }
}
```

What is the result of executing `Question_5_1.UP.printValue()`?
A. `1`

B. `0`
C. Compile-time error
D. Run-time error

2. Given:

```
enum Color {
    Blue, Green, Black
}
public class Question_5_2 {
    public static void main(String[] args) {
        Color c = Color.values()[0];
        switch(c) {
            case Blue: System.out.println(1); break;
            case Green: System.out.println(2); break;
            case Black: System.out.println(3); break;
        }
    }
}
```

What is the result of executing `Question_5_2` ?
A. `1`
B. `2`
C. `3`
D. Compile-time error
E. Run-time error

3. Which of the following statements is true?
A. Enums are thread-safe.
B. Enums can neither extend from a class nor implement an interface.
C. Enums cannot define constructors.
D. Enums cannot have setter methods.

4. Given:

```
enum Level {
    HIGH(100), LOW(10);
    private int value;
    private Level(int value) {
        this.value = value;
        System.out.println(value);
    }
}
public class Question_5_4 {
    public static void main(String[] args) {
        Level l1 = Level.HIGH;
        Level l2 = Level.HIGH;
    }
}
```

What is the result of executing class `Question_5_4` ?
A. `100`
B.
```
100
100
```
C.
```
100
10
```
D. Compile-time error
E. Nothing is printed

5. Given:

```
enum Color1 {
    RED, YELLOW
}
enum Color2 {
    RED, PINK
}
public class Question_5_5 {
    public static void main(String[] args) {
        if(Color1.RED.equals(Color2.RED)) {
            System.out.println(1);
        } else {
            System.out.println(0);
        }
    }
}
```

What is the result of executing `Question_5_5` ?
A. `1`
B. `0`
D. Compile-time error
E. Run-time error

6. Which of the following statements are true?
A. You can compare two enumerations using the `==` operator.
B. Enums implicitly inherit from `java.lang.Enum` .
C. You can't use the `new` operator inside an `enum` .
D. You can't have `abstract` methods inside an `enum` .

Open answers page

04. Interfaces 06. Generics